Java Stream API Notes

Introduction

First of all, Java 8 Streams should not be confused with Java I/O streams (ex: FileInputStream etc); these have very little to do with each other.

Simply put, streams are wrappers around a data source, allowing us to operate with that data source and making bulk processing convenient and fast.

A stream does not store data and, in that sense, is not a data structure. It also never modifies the underlying data source.

This functionality – java.util.stream – supports functional-style operations on streams of elements, such as map-reduce transformations on collections.

Let's now dive into few simple examples of stream creation and usage – before getting into terminology and core concepts.

Java Stream Creation

Let's first obtain a stream from an existing array:

```
private static Employee[] arrayOfEmps = {
   new Employee(1, "Jeff Bezos", 100000.0),
   new Employee(2, "Bill Gates", 200000.0),
   new Employee(3, "Mark Zuckerberg", 300000.0)
};


Stream.of(arrayOfEmps);
```

We can also obtain a stream from an existing list:

```
private static List<Employee> empList = Arrays.asList(arrayOfEmps);

empList.stream();
```

Note that Java 8 added a new stream() method to the Collection interface.

And we can create a stream from individual objects using Stream.of():

```
Stream.of(arrayOfEmps[0], arrayOfEmps[1], arrayOfEmps[2]);
```

Or simply using Stream.builder():

```
Stream.Builder<Employee> empStreamBuilder = Stream.builder();


empStreamBuilder.accept(arrayOfEmps[0]);
```

```
empStreamBuilder.accept(arrayOfEmps[1]);
empStreamBuilder.accept(arrayOfEmps[2]);


Stream<Employee> empStream = empStreamBuilder.build();
```

There are also other ways to obtain a stream, some of which we will see in sections below.

Java Stream Operations

Let's now see some common usages and operations we can perform on and with the help of the stream support in the language.

forEach

forEach() is simplest and most common operation; it loops over the stream elements, calling the supplied function on each element.

The method is so common that is has been introduced directly in Iterable, Map etc:

```
@Test
public void whenIncrementSalaryForEachEmployee_thenApplyNewSalary() {
   empList.stream().forEach(e -> e.salaryIncrement(10.0));


   assertThat(empList, contains(
     hasProperty("salary", equalTo(110000.0)),
     hasProperty("salary", equalTo(220000.0)),
     hasProperty("salary", equalTo(330000.0))
   ));
}
```

This will effectively call the salaryIncrement() on each element in the empList.

forEach() is a terminal operation, which means that, after the operation is performed, the stream pipeline is considered consumed, and can no longer be used. We'll talk more about terminal operations in the next section.

map

map() produces a new stream after applying a function to each element of the original stream. The new stream could be of different type.

The following example converts the stream of Integers into the stream of Employees:

Java Stream API Notes

```
@Test
public void whenMapIdToEmployees_thenGetEmployeeStream() {
    Integer[] empIds = { 1, 2, 3 };

    List<Employee> employees = Stream.of(empIds)
      .map(employeeRepository::findById)
      .collect(Collectors.toList());

    assertEquals(employees.size(), empIds.length);
}
```

Here, we obtain an Integer stream of employee ids from an array. Each Integer is passed to the function employeeRepository::findById() – which returns the corresponding Employee object; this effectively forms an Employee stream.

collect

We saw how collect() works in the previous example; its one of the common ways to get stuff out of the stream once we are done with all the processing:

```
@Test
public void whenCollectStreamToList_thenGetList() {
    List<Employee> employees = empList.stream().collect(Collectors.toList());

    assertEquals(empList, employees);
}
```

collect() performs mutable fold operations (repackaging elements to some data structures and applying some additional logic, concatenating them, etc.) on data elements held in the Stream instance.

The strategy for this operation is provided via the Collector interface implementation. In the example above, we used the toList collector to collect all Stream elements into a List instance.

filter

Next, let's have a look at filter(); this produces a new stream that contains elements of the original stream that pass a given test (specified by a Predicate).

Let's have a look at how that works:

```
@Test
```

```
public void whenFilterEmployees_thenGetFilteredStream() {
    Integer[] empIds = { 1, 2, 3, 4 };

    List<Employee> employees = Stream.of(empIds)
      .map(employeeRepository::findById)
      .filter(e -> e != null)
      .filter(e -> e.getSalary() > 200000)
      .collect(Collectors.toList());

    assertEquals(Arrays.asList(arrayOfEmps[2]), employees);
}
```

In the example above, we first filter out null references for invalid employee ids and then again apply a filter to only keep employees with salaries over a certain threshold.

findFirst

findFirst() returns an Optional for the first entry in the stream; the Optional can, of course, be empty:

```
@Test
public void whenFindFirst_thenGetFirstEmployeeInStream() {
    Integer[] empIds = { 1, 2, 3, 4 };

    Employee employee = Stream.of(empIds)
      .map(employeeRepository::findById)
      .filter(e -> e != null)
      .filter(e -> e.getSalary() > 100000)
      .findFirst()
      .orElse(null);

    assertEquals(employee.getSalary(), new Double(200000));
}
```

Java Stream API Notes

Here, the first employee with the salary greater than 100000 is returned. If no such employee exists, then null is returned.

toArray

We saw how we used collect() to get data out of the stream. If we need to get an array out of the stream, we can simply use toArray():

```
@Test
public void whenStreamToArray_thenGetArray() {
    Employee[] employees = empList.stream().toArray(Employee[]::new);

    assertThat(empList.toArray(), equalTo(employees));
}
```

The syntax Employee[]::new creates an empty array of Employee – which is then filled with elements from the stream.

flatMap

A stream can hold complex data structures like Stream<List<String>>. In cases like this, flatMap() helps us to flatten the data structure to simplify further operations:

```
@Test
public void whenFlatMapEmployeeNames_thenGetNameStream() {
    List<List<String>> namesNested = Arrays.asList(
      Arrays.asList("Jeff", "Bezos"),
      Arrays.asList("Bill", "Gates"),
      Arrays.asList("Mark", "Zuckerberg"));

    List<String> namesFlatStream = namesNested.stream()
      .flatMap(Collection::stream)
      .collect(Collectors.toList());

    assertEquals(namesFlatStream.size(), namesNested.size() * 2);
}
```

Java Stream API Notes

Notice how we were able to convert the Stream<List<String>> to a simpler Stream<String> – using the flatMap() API.

peek

We saw forEach() earlier in this section, which is a terminal operation. However, sometimes we need to perform multiple operations on each element of the stream before any terminal operation is applied.

peek() can be useful in situations like this. Simply put, it performs the specified operation on each element of the stream and returns a new stream which can be used further. peek() is an intermediate operation:

```
@Test
public void whenIncrementSalaryUsingPeek_thenApplyNewSalary() {
    Employee[] arrayOfEmps = {
        new Employee(1, "Jeff Bezos", 100000.0),
        new Employee(2, "Bill Gates", 200000.0),
        new Employee(3, "Mark Zuckerberg", 300000.0)
    };

    List<Employee> empList = Arrays.asList(arrayOfEmps);

    empList.stream()
      .peek(e -> e.salaryIncrement(10.0))
      .peek(System.out::println)
      .collect(Collectors.toList());

    assertThat(empList, contains(
      hasProperty("salary", equalTo(110000.0)),
      hasProperty("salary", equalTo(220000.0)),
      hasProperty("salary", equalTo(330000.0))
    ));
}
```

Java Stream API Notes

Here, the first peek() is used to increment the salary of each employee. The second peek() is used to print the employees. Finally, collect() is used as the terminal operation.

Method Types and Pipelines

As we've been discussing, Java stream operations are divided into intermediate and terminal operations.

Intermediate operations such as filter() return a new stream on which further processing can be done. Terminal operations, such as forEach(), mark the stream as consumed, after which point it can no longer be used further.

A stream pipeline consists of a stream source, followed by zero or more intermediate operations, and a terminal operation.

Here's a sample stream pipeline, where empList is the source, filter() is the intermediate operation and count is the terminal operation:

```
@Test
public void whenStreamCount_thenGetElementCount() {
    Long empCount = empList.stream()
      .filter(e -> e.getSalary() > 200000)
      .count();

    assertEquals(empCount, new Long(1));
}
```

Some operations are deemed short-circuiting operations. Short-circuiting operations allow computations on infinite streams to complete in finite time:

```
@Test
public void whenLimitInfiniteStream_thenGetFiniteElements() {
    Stream<Integer> infiniteStream = Stream.iterate(2, i -> i * 2);

    List<Integer> collect = infiniteStream
      .skip(3)
      .limit(5)
      .collect(Collectors.toList());

    assertEquals(collect, Arrays.asList(16, 32, 64, 128, 256));
```

```
}
```

Here, we use short-circuiting operations skip() to skip first 3 elements, and limit() to limit to 5 elements from the infinite stream generated using iterate().

We'll talk more about infinite streams later on.

Lazy Evaluation

One of the most important characteristics of Java streams is that they allow for significant optimizations through lazy evaluations.

Computation on the source data is only performed when the terminal operation is initiated, and source elements are consumed only as needed.

All intermediate operations are lazy, so they're not executed until a result of a processing is actually needed.

For example, consider the findFirst() example we saw earlier. How many times is the map() operation performed here? 4 times, since the input array contains 4 elements?

```
@Test

public void whenFindFirst_thenGetFirstEmployeeInStream() {

   Integer[] empIds = { 1, 2, 3, 4 };


   Employee employee = Stream.of(empIds)

     .map(employeeRepository::findById)

     .filter(e -> e != null)

     .filter(e -> e.getSalary() > 100000)

     .findFirst()

     .orElse(null);


   assertEquals(employee.getSalary(), new Double(200000));

}
```

Stream performs the map and two filter operations, one element at a time.

It first performs all the operations on id 1. Since the salary of id 1 is not greater than 100000, the processing moves on to the next element.

Id 2 satisfies both of the filter predicates and hence the stream evaluates the terminal operation findFirst() and returns the result.

Java Stream API Notes

No operations are performed on id 3 and 4.

Processing streams lazily allows avoiding examining all the data when that's not necessary. This behavior becomes even more important when the input stream is infinite and not just very large.

Comparison Based Stream Operations

sorted

Let's start with the sorted() operation – this sorts the stream elements based on the comparator passed we pass into it.

For example, we can sort Employees based on their names:

```
@Test
public void whenSortStream_thenGetSortedStream() {
   List<Employee> employees = empList.stream()
     .sorted((e1, e2) -> e1.getName().compareTo(e2.getName()))
     .collect(Collectors.toList());

   assertEquals(employees.get(0).getName(), "Bill Gates");
   assertEquals(employees.get(1).getName(), "Jeff Bezos");
   assertEquals(employees.get(2).getName(), "Mark Zuckerberg");
}
```

Note that short-circuiting will not be applied for sorted().

This means, in the example above, even if we had used findFirst() after the sorted(), the sorting of all the elements is done before applying the findFirst(). This happens because the operation cannot know what the first element is until the entire stream is sorted.

min and max

As the name suggests, min() and max() return the minimum and maximum element in the stream respectively, based on a comparator. They return an Optional since a result may or may not exist (due to, say, filtering):

```
@Test
public void whenFindMin_thenGetMinElementFromStream() {
   Employee firstEmp = empList.stream()
     .min((e1, e2) -> e1.getId() - e2.getId())
     .orElseThrow(NoSuchElementException::new);
```

```
    assertEquals(firstEmp.getId(), new Integer(1));
}
```

We can also avoid defining the comparison logic by using Comparator.comparing():

```
@Test
public void whenFindMax_thenGetMaxElementFromStream() {
    Employee maxSalEmp = empList.stream()
      .max(Comparator.comparing(Employee::getSalary))
      .orElseThrow(NoSuchElementException::new);

    assertEquals(maxSalEmp.getSalary(), new Double(300000.0));
}
```

distinct

distinct() does not take any argument and returns the distinct elements in the stream, eliminating duplicates. It uses the equals() method of the elements to decide whether two elements are equal or not:

```
@Test
public void whenApplyDistinct_thenRemoveDuplicatesFromStream() {
    List<Integer> intList = Arrays.asList(2, 5, 3, 2, 4, 3);
    List<Integer> distinctIntList = intList.stream().distinct().collect(Collectors.toList());

    assertEquals(distinctIntList, Arrays.asList(2, 5, 3, 4));
}
```

allMatch, anyMatch, and noneMatch

These operations all take a predicate and return a boolean. Short-circuiting is applied and processing is stopped as soon as the answer is determined:

```
@Test
public void whenApplyMatch_thenReturnBoolean() {
    List<Integer> intList = Arrays.asList(2, 4, 5, 6, 8);
```

```
    boolean allEven = intList.stream().allMatch(i -> i % 2 == 0);

    boolean oneEven = intList.stream().anyMatch(i -> i % 2 == 0);

    boolean noneMultipleOfThree = intList.stream().noneMatch(i -> i % 3 == 0);


    assertEquals(allEven, false);

    assertEquals(oneEven, true);

    assertEquals(noneMultipleOfThree, false);

}
```

allMatch() checks if the predicate is true for all the elements in the stream. Here, it returns false as soon as it encounters 5, which is not divisible by 2.

anyMatch() checks if the predicate is true for any one element in the stream. Here, again short-circuiting is applied and true is returned immediately after the first element.

noneMatch() checks if there are no elements matching the predicate. Here, it simply returns false as soon as it encounters 6, which is divisible by 3.

Java Stream Specializations

From what we discussed so far, Stream is a stream of object references. However, there are also the IntStream, LongStream, and DoubleStream – which are primitive specializations for int, long and double respectively. These are quite convenient when dealing with a lot of numerical primitives.

These specialized streams do not extend Stream but extend BaseStream on top of which Stream is also built.

As a consequence, not all operations supported by Stream are present in these stream implementations. For example, the standard min() and max() take a comparator, whereas the specialized streams do not.

Creation

The most common way of creating an IntStream is to call mapToInt() on an existing stream:

```
@Test

public void whenFindMaxOnIntStream_thenGetMaxInteger() {

    Integer latestEmpId = empList.stream()

      .mapToInt(Employee::getId)

      .max()

      .orElseThrow(NoSuchElementException::new);
```

```
    assertEquals(latestEmpId, new Integer(3));
}
```

Here, we start with a Stream<Employee> and get an IntStream by supplying the Employee::getId to mapToInt. Finally, we call max() which returns the highest integer.

We can also use IntStream.of() for creating the IntStream:

```
IntStream.of(1, 2, 3);
```

or IntStream.range():

```
IntStream.range(10, 20)
```

which creates IntStream of numbers 10 to 19.

One important distinction to note before we move on to the next topic:

```
Stream.of(1, 2, 3)
```

This returns a Stream<Integer> and not IntStream.

Similarly, using map() instead of mapToInt() returns a Stream<Integer> and not an IntStream.:

```
empList.stream().map(Employee::getId);
```

Specialized Operations

Specialized streams provide additional operations as compared to the standard Stream – which are quite convenient when dealing with numbers.

For example sum(), average(), range() etc:

```
@Test
public void whenApplySumOnIntStream_thenGetSum() {
    Double avgSal = empList.stream()
      .mapToDouble(Employee::getSalary)
      .average()
      .orElseThrow(NoSuchElementException::new);

    assertEquals(avgSal, new Double(200000));
}
```

Reduction Operations

Java Stream API Notes

A reduction operation (also called as fold) takes a sequence of input elements and combines them into a single summary result by repeated application of a combining operation. We already saw few reduction operations like findFirst(), min() and max().

Let's see the general-purpose reduce() operation in action.

reduce

The most common form of reduce() is:

T reduce(T identity, BinaryOperator<T> accumulator)

where identity is the starting value and accumulator is the binary operation we repeated apply.

For example:

```
@Test
public void whenApplyReduceOnStream_thenGetValue() {
   Double sumSal = empList.stream()
     .map(Employee::getSalary)
     .reduce(0.0, Double::sum);


   assertEquals(sumSal, new Double(600000));
}
```

Here, we start with the initial value of 0 and repeated apply Double::sum() on elements of the stream. Effectively we've implemented the DoubleStream.sum() by applying reduce() on Stream.

Advanced collect

We already saw how we used Collectors.toList() to get the list out of the stream. Let's now see few more ways to collect elements from the stream.

joining

```
@Test
public void whenCollectByJoining_thenGetJoinedString() {
   String empNames = empList.stream()
     .map(Employee::getName)
     .collect(Collectors.joining(", "))
     .toString();
```

```
    assertEquals(empNames, "Jeff Bezos, Bill Gates, Mark Zuckerberg");
}
```

Collectors.joining() will insert the delimiter between the two String elements of the stream. It internally uses a java.util.StringJoiner to perform the joining operation.

toSet

We can also use toSet() to get a set out of stream elements:

```
@Test
public void whenCollectBySet_thenGetSet() {
    Set<String> empNames = empList.stream()
        .map(Employee::getName)
        .collect(Collectors.toSet());

    assertEquals(empNames.size(), 3);
}
```

toCollection

We can use Collectors.toCollection() to extract the elements into any other collection by passing in a Supplier<Collection>. We can also use a constructor reference for the Supplier:

```
@Test
public void whenToVectorCollection_thenGetVector() {
    Vector<String> empNames = empList.stream()
        .map(Employee::getName)
        .collect(Collectors.toCollection(Vector::new));

    assertEquals(empNames.size(), 3);
}
```

Here, an empty collection is created internally, and its add() method is called on each element of the stream.

summarizingDouble

summarizingDouble() is another interesting collector – which applies a double-producing mapping function to each input element and returns a special class containing statistical information for the resulting values:

```
@Test
public void whenApplySummarizing_thenGetBasicStats() {
    DoubleSummaryStatistics stats = empList.stream()
      .collect(Collectors.summarizingDouble(Employee::getSalary));

    assertEquals(stats.getCount(), 3);
    assertEquals(stats.getSum(), 600000.0, 0);
    assertEquals(stats.getMin(), 100000.0, 0);
    assertEquals(stats.getMax(), 300000.0, 0);
    assertEquals(stats.getAverage(), 200000.0, 0);
}
```

Notice how we can analyze the salary of each employee and get statistical information on that data – such as min, max, average etc.

summaryStatistics() can be used to generate similar result when we're using one of the specialized streams:

```
@Test
public void whenApplySummaryStatistics_thenGetBasicStats() {
    DoubleSummaryStatistics stats = empList.stream()
      .mapToDouble(Employee::getSalary)
      .summaryStatistics();

    assertEquals(stats.getCount(), 3);
    assertEquals(stats.getSum(), 600000.0, 0);
    assertEquals(stats.getMin(), 100000.0, 0);
    assertEquals(stats.getMax(), 300000.0, 0);
    assertEquals(stats.getAverage(), 200000.0, 0);
}
```

Java Stream API Notes

partitioningBy

We can partition a stream into two – based on whether the elements satisfy certain criteria or not.

Let's split our List of numerical data, into even and ods:

```
@Test
public void whenStreamPartition_thenGetMap() {
   List<Integer> intList = Arrays.asList(2, 4, 5, 6, 8);
   Map<Boolean, List<Integer>> isEven = intList.stream().collect(
     Collectors.partitioningBy(i -> i % 2 == 0));

   assertEquals(isEven.get(true).size(), 4);
   assertEquals(isEven.get(false).size(), 1);
}
```

Here, the stream is partitioned into a Map, with even and odds stored as true and false keys.

groupingBy

groupingBy() offers advanced partitioning – where we can partition the stream into more than just two groups.

It takes a classification function as its parameter. This classification function is applied to each element of the stream.

The value returned by the function is used as a key to the map that we get from the groupingBy collector:

```
@Test
public void whenStreamGroupingBy_thenGetMap() {
   Map<Character, List<Employee>> groupByAlphabet = empList.stream().collect(
     Collectors.groupingBy(e -> new Character(e.getName().charAt(0))));

   assertEquals(groupByAlphabet.get('B').get(0).getName(), "Bill Gates");
   assertEquals(groupByAlphabet.get('J').get(0).getName(), "Jeff Bezos");
   assertEquals(groupByAlphabet.get('M').get(0).getName(), "Mark Zuckerberg");
}
```

In this quick example, we grouped the employees based on the initial character of their first name.

Java Stream API Notes

mapping

groupingBy() discussed in the section above, groups elements of the stream with the use of a Map.

However, sometimes we might need to group data into a type other than the element type.

Here's how we can do that; we can use mapping() which can actually adapt the collector to a different type – using a mapping function:

```
@Test
public void whenStreamMapping_thenGetMap() {
   Map<Character, List<Integer>> idGroupedByAlphabet = empList.stream().collect(
     Collectors.groupingBy(e -> new Character(e.getName().charAt(0)),
       Collectors.mapping(Employee::getId, Collectors.toList())));

   assertEquals(idGroupedByAlphabet.get('B').get(0), new Integer(2));
   assertEquals(idGroupedByAlphabet.get('J').get(0), new Integer(1));
   assertEquals(idGroupedByAlphabet.get('M').get(0), new Integer(3));
}
```

Here mapping() maps the stream element Employee into just the employee id – which is an Integer – using the getId() mapping function. These ids are still grouped based on the initial character of employee first name.

reducing

reducing() is similar to reduce() – which we explored before. It simply returns a collector which performs a reduction of its input elements:

```
@Test
public void whenStreamReducing_thenGetValue() {
   Double percentage = 10.0;
   Double salIncrOverhead = empList.stream().collect(Collectors.reducing(
      0.0, e -> e.getSalary() * percentage / 100, (s1, s2) -> s1 + s2));

   assertEquals(salIncrOverhead, 60000.0, 0);
}
```

Here reducing() gets the salary increment of each employee and returns the sum.

Java Stream API Notes

reducing() is most useful when used in a multi-level reduction, downstream of groupingBy() or partitioningBy(). To perform a simple reduction on a stream, use reduce() instead.

For example, let's see how we can use reducing() with groupingBy():

```
@Test
public void whenStreamGroupingAndReducing_thenGetMap() {
   Comparator<Employee> byNameLength = Comparator.comparing(Employee::getName);

   Map<Character, Optional<Employee>> longestNameByAlphabet = empList.stream().collect(
     Collectors.groupingBy(e -> new Character(e.getName().charAt(0)),
       Collectors.reducing(BinaryOperator.maxBy(byNameLength))));

   assertEquals(longestNameByAlphabet.get('B').get().getName(), "Bill Gates");
   assertEquals(longestNameByAlphabet.get('J').get().getName(), "Jeff Bezos");
   assertEquals(longestNameByAlphabet.get('M').get().getName(), "Mark Zuckerberg");
}
```

Here we group the employees based on the initial character of their first name. Within each group, we find the employee with the longest name.

Parallel Streams

Using the support for parallel streams, we can perform stream operations in parallel without having to write any boilerplate code; we just have to designate the stream as parallel:

```
@Test
public void whenParallelStream_thenPerformOperationsInParallel() {
   Employee[] arrayOfEmps = {
     new Employee(1, "Jeff Bezos", 100000.0),
     new Employee(2, "Bill Gates", 200000.0),
     new Employee(3, "Mark Zuckerberg", 300000.0)
   };

   List<Employee> empList = Arrays.asList(arrayOfEmps);
```

```
    empList.stream().parallel().forEach(e -> e.salaryIncrement(10.0));


    assertThat(empList, contains(
      hasProperty("salary", equalTo(110000.0)),
      hasProperty("salary", equalTo(220000.0)),
      hasProperty("salary", equalTo(330000.0))
    ));
}
```

Here salaryIncrement() would get executed in parallel on multiple elements of the stream, by simply adding the parallel() syntax.

This functionality can, of course, be tuned and configured further, if you need more control over the performance characteristics of the operation.

As is the case with writing multi-threaded code, we need to be aware of few things while using parallel streams:

1. We need to ensure that the code is thread-safe. Special care needs to be taken if the operations performed in parallel modifies shared data.

2. We should not use parallel streams if the order in which operations are performed or the order returned in the output stream matters. For example operations like findFirst() may generate the different result in case of parallel streams.

3. Also, we should ensure that it is worth making the code execute in parallel. Understanding the performance characteristics of the operation in particular, but also of the system as a whole – is naturally very important here.

Infinite Streams

Sometimes, we might want to perform operations while the elements are still getting generated. We might not know beforehand how many elements we'll need. Unlike using list or map, where all the elements are already populated, we can use infinite streams, also called as unbounded streams.

There are two ways to generate infinite streams:

generate

We provide a Supplier to generate() which gets called whenever new stream elements need to be generated:

```
@Test
public void whenGenerateStream_thenGetInfiniteStream() {
    Stream.generate(Math::random)
```

```
    .limit(5)
    .forEach(System.out::println);
}
```

Here, we pass Math::random() as a Supplier, which returns the next random number.

With infinite streams, we need to provide a condition to eventually terminate the processing. One common way of doing this is using limit(). In above example, we limit the stream to 5 random numbers and print them as they get generated.

Please note that the Supplier passed to generate() could be stateful and such stream may not produce the same result when used in parallel.

iterate

iterate() takes two parameters: an initial value, called seed element and a function which generates next element using the previous value. iterate(), by design, is stateful and hence may not be useful in parallel streams:

```
@Test
public void whenIterateStream_thenGetInfiniteStream() {
   Stream<Integer> evenNumStream = Stream.iterate(2, i -> i * 2);

   List<Integer> collect = evenNumStream
     .limit(5)
     .collect(Collectors.toList());

   assertEquals(collect, Arrays.asList(2, 4, 8, 16, 32));
}
```

Here, we pass 2 as the seed value, which becomes the first element of our stream. This value is passed as input to the lambda, which returns 4. This value, in turn, is passed as input in the next iteration.

This continues until we generate the number of elements specified by limit() which acts as the terminating condition.

File Operations

Let's see how we could use the stream in file operations.

File Write Operation

```
@Test
```

Java Stream API Notes

```
public void whenStreamToFile_thenGetFile() throws IOException {
    String[] words = {
      "hello",
      "refer",
      "world",
      "level"
    };

    try (PrintWriter pw = new PrintWriter(
      Files.newBufferedWriter(Paths.get(fileName)))) {
        Stream.of(words).forEach(pw::println);
    }
}
```

Here we use forEach() to write each element of the stream into the file by calling PrintWriter.println().

File Read Operation

```
private List<String> getPalindrome(Stream<String> stream, int length) {
    return stream.filter(s -> s.length() == length)
      .filter(s -> s.compareToIgnoreCase(
        new StringBuilder(s).reverse().toString()) == 0)
      .collect(Collectors.toList());
}


@Test
public void whenFileToStream_thenGetStream() throws IOException {
    List<String> str = getPalindrome(Files.lines(Paths.get(fileName)), 5);
    assertThat(str, contains("refer", "level"));
}
```

Here Files.lines() returns the lines from the file as a Stream which is consumed by the getPalindrome() for further processing.

Java Stream API Notes

getPalindrome() works on the stream, completely unaware of how the stream was generated. This also increases code reusability and simplifies unit testing.

Java Streams Improvements In Java 9

Java 8 brought Java streams to the world. However, the following version of the language also contributed to the feature. So, we'll now give a brief overview of the improvements that Java 9 brought to the Streams API. Let's do it.

takeWhile

The takeWhile method is one of the new additions to the Streams API. It does what its name implies: it takes (elements from a stream) while a given condition is true. The moment the condition becomes false, it quits and returns a new stream with just the elements that matched the predicate. In other words, it's like a filter with a condition. Let's see a quick example.

```
Stream.iterate(1, i -> i + 1)
        .takeWhile(n -> n <= 10)
        .map(x -> x * x)
        .forEach(System.out::println);
```

In the code above we obtain an infinite stream and then use the takeWhile method to select the numbers that are less than or equals to 10. After that, we calculate their squares and print those.

You might be wondering what's the difference between takeWhile and filter. After all, you could accomplish the same result with the following code:

```
Stream.iterate(1, i -> i + 1)
        .filter(x -> x <= 10)
        .map(x -> x * x)
        .forEach(System.out::println);
```

Well, in this particular scenario, the two methods achieve the same result, but that's not always the case. Let's illustrate the difference with another example:

```
Stream.of(1,2,3,4,5,6,7,8,9,0,9,8,7,6,5,4,3,2,1,0)
        .takeWhile(x -> x <= 5)
        .forEach(System.out::println);


Stream.of(1,2,3,4,5,6,7,8,9,0,9,8,7,6,5,4,3,2,1,0)
        .filter(x -> x <= 5)
        .forEach(System.out::println);
```

Java Stream API Notes

Here, we have two identical streams, which we filter using takeWhile and filter, respectively. So, what's the difference? If you run the code above you'll see that the first version prints out:

1

2

3

4

5

while the version with filter results in

1

2

3

4

5

0

5

4

3

2

1

0

As you can see, filter() applies the predicate throughout the whole sequence. On the other hand, takeWhile stops evaluating as soon as it finds the first occurrence where the condition is false.

dropWhile

The dropWhile method does pretty much the same thing the takewhile does but in reverse. Confused? It's simple: while takewhile takes while its condition is true, dropwhile drops elements while the condition is true. That is to say: the previous method uses the predicate (the condition) to select the elements to preserve in the new stream it returns. This method does the opposite, using the condition to select the items not to include in the resulting stream. Let's see an example:

```
Stream.of(1,2,3,4,5,6,7,8,9,0,9,8,7,6,5,4,3,2,1,0)
    .dropWhile(x -> x <= 5)
    .forEach(System.out::println);
```

This is the same as the previous example, the only difference being that we're using dropWhile instead of takeWhile. That is to say, we're now dropping elements that are less than or equals to five. The resulting items are:

6

7

8

9

0

9

8

7

6

5

4

3

2

1

0

As you can see, there are numbers less than or equals to five in the latter half of the sequence. Why? It's simple: they came after the first element which failed to match the predicate, so the method stopped dropping at that point.

iterate

We've already mentioned the original iterate() method that was introduced in the 8th version of Java. Java 9 brings an override of the method. So, what's the difference?

As you've learned, the original incarnation of the method had two arguments: the initializer (a.k.a. the seed) and the function that generates the next value. The problem with the method is that it didn't include a way for the loop to quit. That's great when you're trying to create infinite streams, but that's not always the case.

In Java 9 we have the new version of iterate(), which adds a new parameter, which is a predicate used to decide when the loop should terminate. As long as the condition remains true, we keep going.

Consider the following example:

Stream.

```
        iterate(1, i -> i < 256, i -> i * 2)
        .forEach(System.out::println);
```

The code above prints the powers of two, as long as they're less than 256. We could say that the new iterate() method is a replacement for the good-old for statement. In fact, the code above is equivalent to the following excerpt:

```
for (int i = 1; i < 256; i*=2) {
        System.out.println(i);
}
```

ofNullable

The last item in this list of additions to the Stream APIs is a powerful way not only to avoid the dreaded null pointer exception but also to write cleaner code. Hopefully, it's very straightforward. Check out the following example:

```
Stream<Integer> result = number != null
    ? Stream.of(number)
    : Stream.empty();
```

Assume that number refers to some integer obtained through the UI, the network, filesystem or another external untrusted source. So, it could be null. We wouldn't want to create a stream with a null element; that could result in a null pointer exception at some point. To avoid that that we can check for null and return an empty stream.

The example above is a contrived example, sure. In real life, code in similar scenarios could become really messy, really fast. We could employ ofNullable() instead:

```
Stream<Integer> result = Stream.ofNullable(number);
```

The new method returns empty Optionals in it receives null, avoiding runtime errors in scenarios that would normally cause one, like in the following example:

```
Integer number = null;
Stream<Integer> result = Stream.ofNullable(number);
result.map(x -> x * x).forEach(System.out::println);
```

%MCEPASTEBIN%

Java Streams: What Are The Next Steps?

In this article, we focused on the details of the new Stream functionality in Java 8. We saw various operations supported and how lambdas and pipelines can be used to write concise code. We also saw some characteristics of streams like lazy evaluation, parallel and infinite streams. You'll find the sources of the examples over on GitHub.

Java Stream API Notes

Now, what should you do next? Well, there's a lot to explore in your journey to be a better Java developer, so here are a few suggestions.

For starters, you can continue your exploration of the concepts you've seen today with a look at the reactive paradigm, made possible by very similar concepts to the one we discussed here.

Additionally, keep in touch with the Stackify blog. We're always publishing articles that might be of interest to you. You might need to learn more about the main Java frameworks, or how to properly handle exceptions in the language. In today's article, we've covered an important feature that was introduced with Java 8. The language has come a long way since then and you might want to check out more recent developments.

Finally, to be a great developer you can't overlook performance. We have posts that cover from Java performance tuning tips to the main tools you should check about, and a lot more in between.

And speaking of tools, you might want to take a look at the free profiler by Stackify, Prefix. With Prefix, you can monitor both Windows desktop and web applications, reviewing their performance, finding hidden exceptions and solving bugs before they get to production.